



TRANSLATORS

ATHANASIOS ROUDIS 5098

STYLIANOS SIMANTIRAKIS 5127

CONTENTS

	Page Number
1) Method of execution	2
2) Files to check correct operation	2
3) Verbal analyzer	3
4) Editorial analyst	7
5) Intermediate code	37
6)Table of symbols	38
7) Final code	42

1) Method of execution

The compiler file is : cpy.py .To run the compiler you need to open a terminal in the folder where you saved cpy.py and run the following command:

```
\the_path_of_the_file>/python cpy.py nameOfTheFileToTest.cpy
```

The code used to make this command possible is shown in Figure 1.

```
1490  #Gia to diabasma apo ta arguments
1491  parser = argparse.ArgumentParser(description='Compile a file.')
1492  parser.add_argument('input_file', help='Path to the input file')
1493
1494  args = parser.parse_args()
1495
1496  fileToCompile = args.input_file
1497
1498  try:
1499      f = open(fileToCompile)
1500  except Exception as e:
1501      print(e)
```

Picture 1

THE intermediate code is produced in the file intermediate-for-(nameOfTheFileToTest.cpy).int and the symbol table is generated in the file symbol-table-for-(nameOfTheFileToTest.cpy).sym .The final code is generated in the file assembly-for-(nameOfTheFileToTest.cpy).asm.

2) Files to check correct operation

The files for the health check are:

- i) test.cpy , the given file to test.
- ii) onlyMainTest.cpy , contains a program with only the main function.
- iii) moreThanOneDeclarationsTest.cpy, contains a program with more than one declarations.
- iv) limitsTest.cpy , contains a program with limit values.
- v)smallTest.cpy, contains a program from the slides.
- vi)ifWhileTest.cpy , contains a program from the slides .
- vii) finalCodeExampleTest.cpy , contains a program from the slides .

3) Verbal analyzer

To create the verbal analyzer, the automaton of Figures 2,3,4 & 5 was used. More specifically, the parser reads the characters from the input file one by one and decides which state of the automaton it is in.

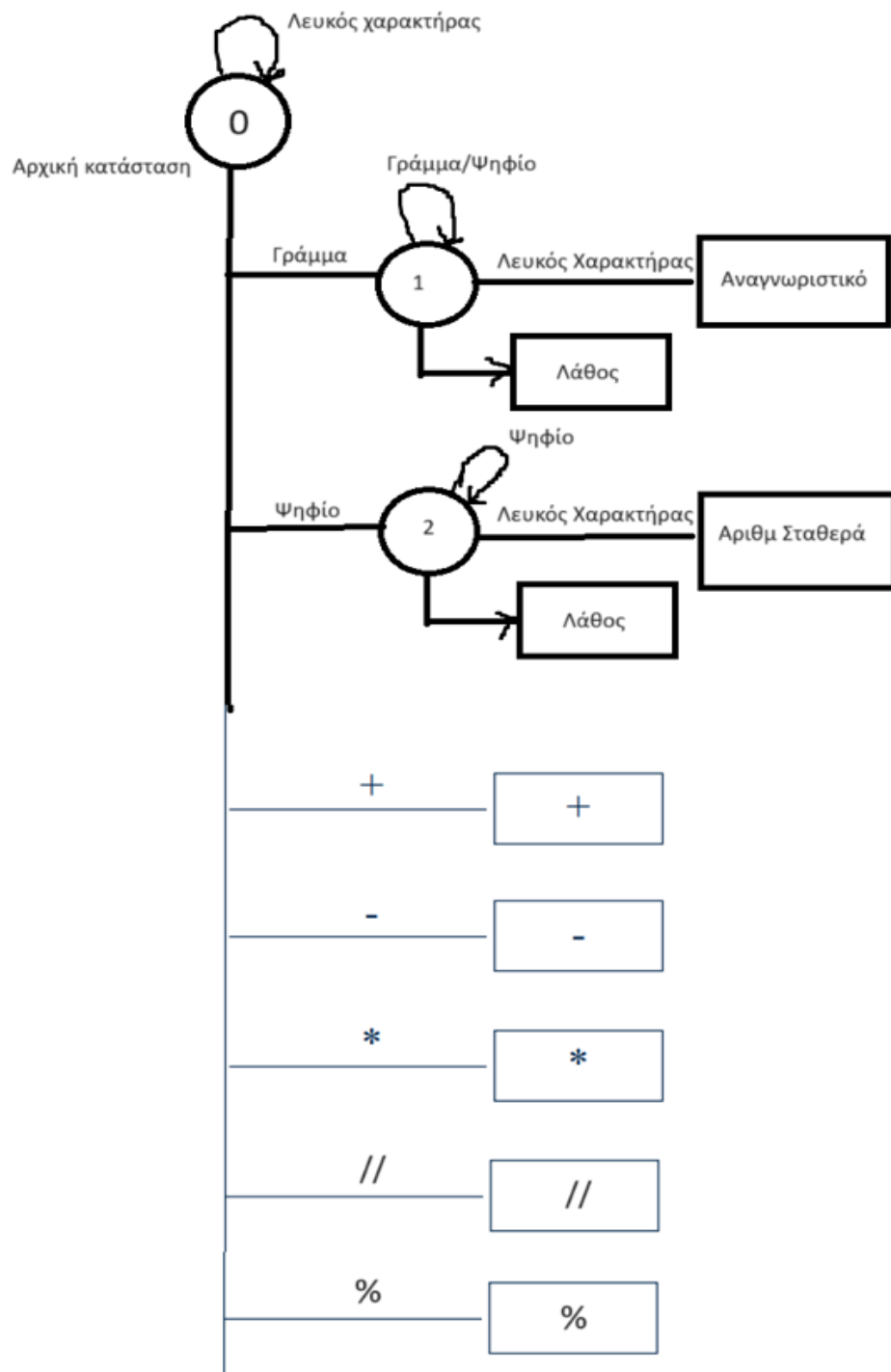


Figure 2

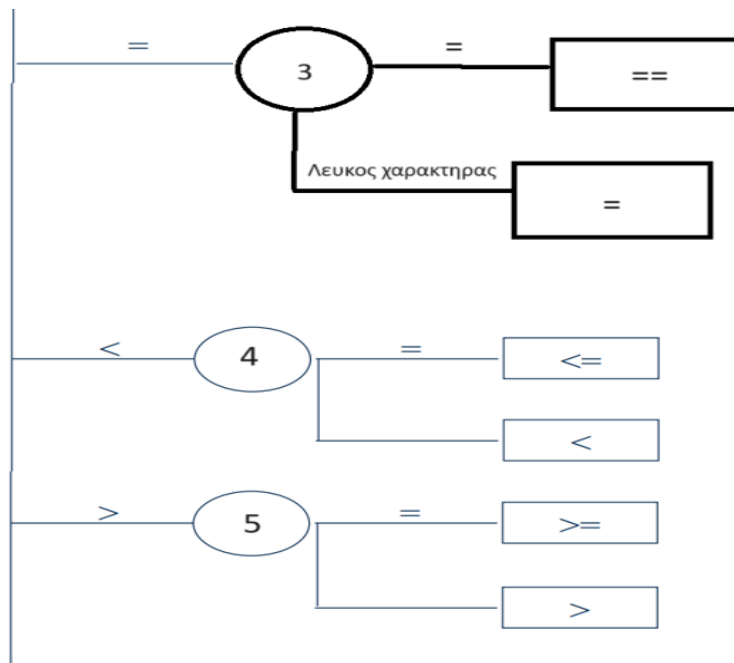


Figure 3

Also at this stage the check is made for the limit values given in the pronunciation. Finally, when an error is detected either by the automatic or by the limit values, the corresponding error message is printed and compilation is terminated.

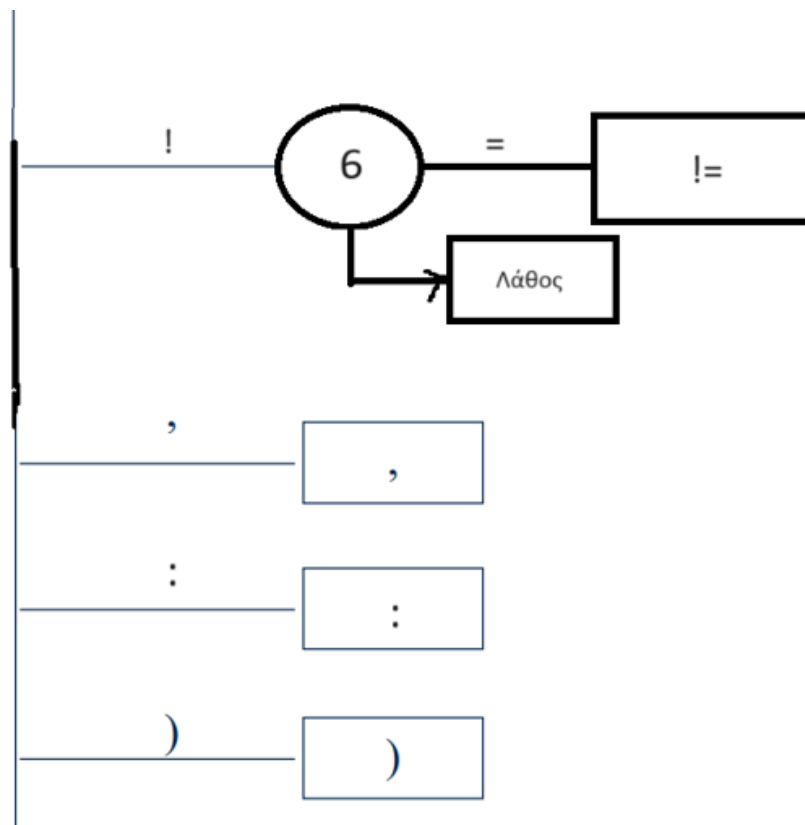


Figure 4

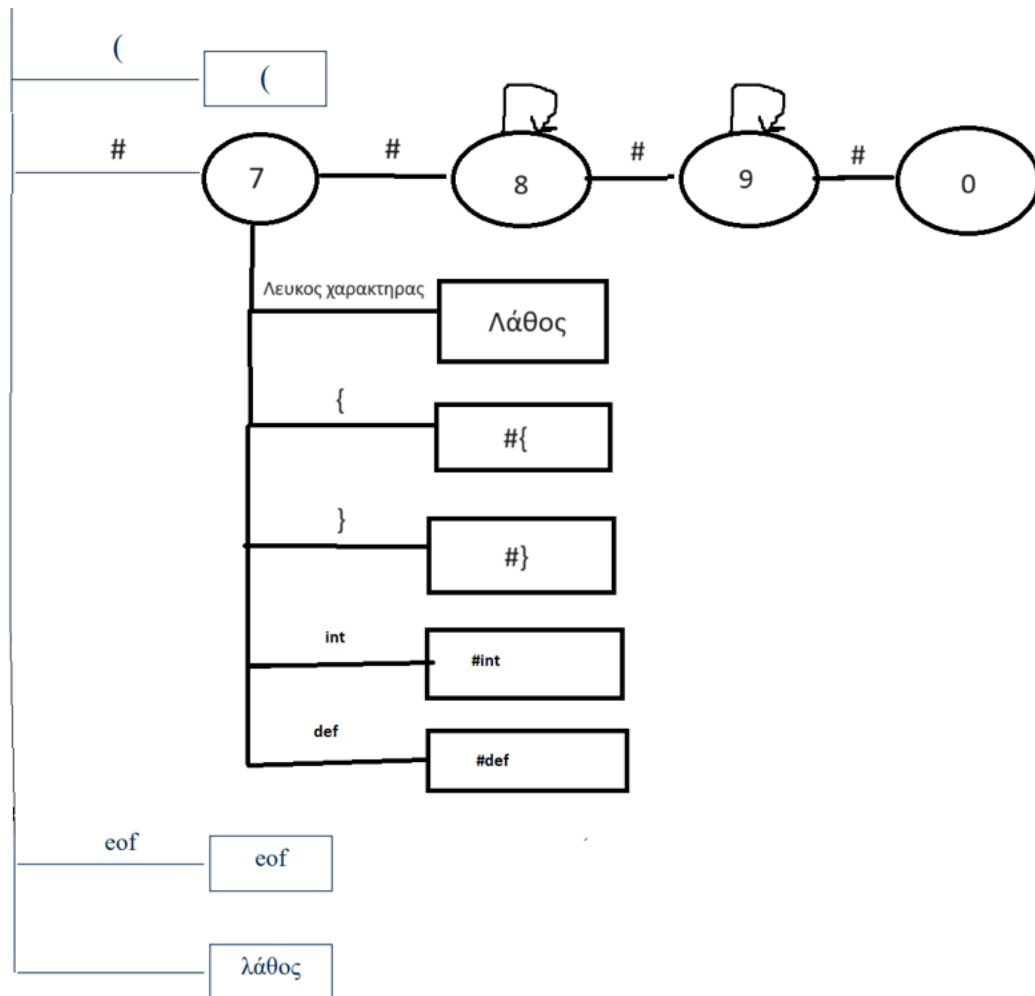


Figure 5

Machine explanation/code:

Situations

- **Status 0:** Original state. It reads the current character and decides the next state based on that character.
 - o If the character is a blank, newline, carriage return, tab, or vertical tab, it remains in state 0.
 - o If the character is a letter, it goes to state 1. If the
 - o character is a digit, it goes to state 2. If the character is=,
 - o goes to state 3. If the character is<, goes to state 4. If the
 - o character is>, goes to state 5. If the character is!, goes to
 - o state 6.
 - o

- o If the character is#, goes to state 7.
 - o If the character is+, returns the token["addtoken", line]. If the
 - o character is-, returns the token["subtoken", line]. If the character
 - o is*, returns the token["multitoken", line]. If the character is/, goes
 - o to state 14.
 - o If the character is%, returns the token["modtoken", line]. If the
 - o character is,, returns the token["commatoken", line].
 - o If the character is:, returns the token["anwkatwtoken", line].
 - o If the character is(, returns the token["leftpartoken", line].
 - o If the character is), returns the token["rightpartoken", line].
 - o For any other character, it displays an error message and stops execution.
- **Situation 1:** Collects words. If the character is a letter or digit, adds it to the word. Otherwise, it returns the appropriate token.
 - o If the word exists incommandList, returns the token ["commandtoken", line, word].
 - o If the word is a unique identifier (up to 30 characters), returns the token["anagnotistikotoken", line, word].
- **Situation 2:** Collects numbers. If the character is a digit, adds it to the number. Otherwise, it returns the token["numbertoken", line, int(number)]if the number is within limits.
- **Situation 3:** Checks if the next character is=.
 - o If it is, it returns the token["isothtatoken", line]. Otherwise,
 - o it returns the token["ana8eshtoken", line].
- **Status 4:** Checks if the next character is=.
 - o If it is, it returns the token["mikistoken", line]. Otherwise, it
 - o returns the token["mikroterotoken", line].
- **Status 5:** Checks if the next character is=.
 - o If it is, it returns the token["megisotoken", line]. Otherwise, it
 - o returns the token["megaluterotoken", line].
- **Status 6:** Checks if the next character is=.
 - o If it is, it returns the token["diaforotoken", line]. Otherwise, it
 - o displays an error message and stops execution.
- **Status 7:** Checks for comments and symbols{,}or recognize special tokens#i(forint) and#d(fordef).
- **Status 8:** Manages comments.
- **Situation 9-12:** Check for special words#intand#def.
- **Status 13:** Continues comments.
- **Status 14:** Checks for//(integer division).

If the function reaches the end of the file, it returns the token["EOFtoken", line].

4) Editorial analyst

Its grammar was used to implement the syntax analyzer

language cpy . Her grammar follows cpy which also gives the exact description

of the language:

```
startRule
:
    declarations
    def_function
    call_main_part
;

def_function
: 'def' ID ('(' id_list ')') ':'
    '#{'
        Declarations
        (def_function)*
        globals
        statements
    '#}'
;

declarations
: (declaration_line)*
;

declaration_line
: '#int' id_list
;

globals
: (globals_line)*
;

globals_line
: 'global' id_list
;

statement
: simple_statement
| structured_statement
;
```



```

statements
    :statement+
    ;

simple_statement
    :assignment_stat
    | print_stat
    | return_stat
    ;

structured_statement
    :if_stat
    | while_stat
    ;

assignment_stat
    :ID'= '
        (expression
        | 'int' '(' 'input' '(' ' ' ')' ')'
        )
    ;

print_stat
    : 'print' '(' 'expression' ')' ;

return_stat
    : 'return' expression
    ;

if_stat
    : 'if' condition ':'
        (statement
        )
        ('elif' condition ':'
        (statement
        )
        )
        ('else:'
        (statement
        )
        )
    ;

while_stat
    : 'while' condition ':'
        (statement
        )
    ;

```

```

: whilecondition':'
  '#{
    (statement
  )
;  '#}'

id_list
: ID('ID)*
|
;

expression
: optional_sign term
  (ADD_OP term)*
;

term
: factor
  (MUL_OP factor)*
;

Factor
: INTEGER
| '(' expression ')'
| idtail ID
;

idtail
: '('actual_par_list'
|
;

actual_par_list
: expression('expression)*
|
;

Optional_sign
: ADD_OP
|
;

condition
:bool_term(orbool_term)*;

```

```

bool_term
    :bool_factor(andbool_factor)*
    ;

bool_factor
    : 'not' '['condition']' | '['condition']'

    | expression REL_OP expression
    ;

call_main_part:
    '#def main'
    declarations
    statements
    ;

```

The parser in each case reads the parser an array containing the token, the line, and in some cases anything else that is useful. When an error is detected, the corresponding error message is printed and compilation is terminated.

Grammar implementation in python:

if_stat rule:

• • Check if the token is command token and if the command is if then if:

- If so, it continues processing.
- Otherwise, the function returns.

• Condition Checking and Reading:

- It calls the function `isCondition(token)` to check if the next token is a condition.
- If the condition is valid, it saves the `listsconditionTrue` and `conditionFalse`.
- If not, it displays an error message and terminates execution.

• Backpatch and Read Next Token:

- Updates `listsconditionTrue` with the next quartet.
- Reads the next token.

• Check for: After the if then if:

- If the token is `anwkatwtoken` (ie:), continues.

- If not, it displays an error message and terminates execution.
- **Checking and Execution of Declaration:**
 - If the statement is valid (isStatement(token)), continues.
 - Creates a listifList with the next four and adds one quartetjump.
 - Updates the listconditionFalse with the next quartet.
 - Saves the file location and reads the next token.
- **Check for Next Command:**
 - If the next command is elif, retroactively calls herifState(token).
 - If the next command is else, calls herelseState(token).
 - If it's any other command, it updates the listifList and returns the location of the file.

```

404 def ifState(token):
405     if token[0] == "commandtoken":
406         if token[2] == "if" or token[2] == "elif":
407             token = lex()
408             cond = isCondition(token)
409             if cond[0]:
410                 conditionTrue = cond[1]
411                 conditionFalse = cond[2]
412                 backpatch(conditionTrue, nextQuad())
413                 token = lex()
414             if token[0] == "anwkatwtoken":
415                 token = lex()
416                 if isStatement(token):
417                     ifList = makeList(nextQuad())
418                     quadList.append(genQuad("jump", "_", "_", "_"))
419                     backpatch(conditionFalse, nextQuad())
420                     seekIndex = f.tell()
421                     token = lex()
422                 if token[0] == "commandtoken":
423                     if token[2] == "elif":
424                         x = ifState(token)
425                         backpatch(ifList, nextQuad())
426                         return x
427                     elif token[2] == "else":
428                         x = elseState(token)
429                         backpatch(ifList, nextQuad())
430                         return x
431                     else:
432                         backpatch(ifList, nextQuad())
433                         f.seek(seekIndex)
434                         return True
435                 else:
436                     backpatch(ifList, nextQuad())
437                     f.seek(seekIndex)
438                     return True
439             else:
440                 print ("Inside the 'if' at line:", token[1], "a statement was expected")
441                 print("Compilation failed")
442                 exit()
443         else:
444             print ("After the 'if' at line:", token[1], "a ':' was expected")
445             print("Compilation failed")
446             exit()
447     else:
448         print ("After the 'if' at line:", token[1], "a condition was expected")
449         print("Compilation failed")
450         exit()
451 
```

Figure 6

- **Check If Command Iselse:**
 - The function starts by verifying that the passed token is the commandelse.
 - If it is, it reads the next token with the functionlex().
- **Control For:After theelse:**
 - If the next token isanwkatwtoken(ie:), moves to the next line.
 - If not, it displays an error message and terminates execution.
- **Checking and Execution of Declaration:**
 - Reads the next token and checks if it is a valid statement using the functionisStatement(token).
 - If the statement is valid, it returnsTrue.
 - If not, it displays an error message and terminates execution.
- **Errors and Exceptions:**
 - If any problem is found (like missing:or invalid statement), displays an error message and terminates execution.

```

452
453     def elseState(token):
454         if token[2] == "else":
455             token = lex()
456             if token[0] == "anwkatwtoken":
457                 token = lex()
458                 if isStatement(token):
459                     return True
460                 else:
461                     print ("Inside the 'else' at line:", token[1], "a statement was expected")
462                     print("Compilation failed")
463                     exit()
464             else:
465                 print ("After the 'else' at line:", token[1], "a ':' was expected")
466                 print("Compilation failed")
467                 exit()
468

```

Figure 7

Condition rule:

- **Initialize Lists:**
 - The listsBtrueandBfalseare initialized as empty.

- **Check and Read Logical Term:**

- It calls the function `isBoolTerm(token)` to check if the next token is a logical term.
- If the boolean is valid, it stores the lists `Q1true` and `Q1false`.
- If not, it displays an error message and terminates execution.

- **Initialize Lists `Btrue` and `Bfalse`:**

- The lists `Btrue` and `Bfalse` are initialized with the values of `Q1true` and `Q1false` respectively.

- **Repeat for the Doeror:**

- Saves the file location and reads the next token.
- If the token is the command `or`, continues processing.
- It calls the function `isBoolTerm(token)` to check the next logic term after the `or`.
- If the logical condition is not valid, it displays an error message and terminates execution.
- Updates lists `Btrue` and `Bfalse` with the new prices.

- **Return Result:**

- If a valid boolean is found, it returns a list of three values: `[True, Btrue, Bfalse]`.
- If not, it displays an error message and terminates execution.

```

468
469 def isCondition(token):
470     Btrue = []
471     Bfalse = []
472     global seekIndex
473     bterm1 = isBoolTerm(token)
474     if bterm1[0]:
475         Q1true = bterm1[1]
476         Q1false = bterm1[2]
477         Btrue = Q1true
478         Bfalse = Q1false
479         seekIndex = f.tell()
480         token = lex()
481         if token[0] == "commandtoken":
482             while token[2] == "or":
483                 backpatch(Bfalse, nextQuad())
484                 seekIndex = f.tell()
485                 token = lex()
486                 bterm2 = isBoolTerm(token)
487                 if bterm2[0] == False:
488                     print("After the 'or' at line:", token[1], "a boolean term was expected")
489                     print("Compilation failed")
490                     exit()
491                 else:
492                     Q2true = bterm2[1]
493                     Q2false = bterm2[2]
494                     Btrue = mergeList(Btrue, Q2true)
495                     Bfalse = Q2false
496                     f.seek(seekIndex)
497                     return [True, Btrue, Bfalse]
498             else:
499                 f.seek(seekIndex)
500                 return [True, Btrue, Bfalse]
501         else:
502             print("At line:", token[1], "a boolean term was expected")
503             print("Compilation failed")
504             exit()

```

Figure 8

boolTerm rule:

- **Initializing Variables:**

- Initializes the lists `Qtrue` and `Qfalse` as blanks.

- **Control and Analysis of the First Logical Factor:**

- It calls the function `isBoolFactor(token)` to check the first logic factor.
- If the first factor is valid, it stores the lists `R1 true` and `R1 false`.
- If invalid, it displays an error message and terminates execution.

- **Processing of the Terms with the Operator `and`:**

- If the first factor is valid, it checks whether the operator follows `and`.
- If there is the `and`, returns to the next logical factor.
- It repeats this process until the operator no longer exists `and`.
- Returns lists `Qtrue` and `Qfalse` after processing.

- **Return of Results:**

- Returns a list of three values: `[True, Qtrue, Qfalse]`.
- If the first logical factor is not valid, it displays an error message and terminates execution.

```

505
506 def isBoolTerm(token):
507     Qtrue = []
508     Qfalse = []
509     global seekIndex
510     bfactor1 = isBoolFactor(token)
511     if bfactor1[0]:
512         R1true = bfactor1[1]
513         R1false = bfactor1[2]
514         Qtrue = R1true
515         Qfalse = R1false
516         seekIndex = f.tell()
517         token = lex()
518         if token[0] == "commandtoken":
519             while token[2] == "and":
520                 backpatch(Qtrue, nextQuad())
521                 seekIndex = f.tell()
522                 token = lex()
523                 bfactor2 = isBoolFactor(token)
524                 if bfactor2[0]:
525                     R2true = bfactor2[1]
526                     R2false = bfactor2[2]
527                     Qfalse = mergeList(Qfalse, R2false)
528                     Qtrue = R2true
529                     seekIndex = f.tell()
530                     token = lex()
531                     if token[0] != "commandtoken":
532                         break
533                 else:
534                     print("After the 'and' at line:", token[1], "a boolean factor was expected")
535                     print("Compilation failed")
536                     exit()
537             f.seek(seekIndex)
538             return [True, Qtrue, Qfalse]
539         else:
540             f.seek(seekIndex)
541             return [True, Qtrue, Qfalse]
542     else:
543         print("At line:", token[1], "a boolean factor was expected")
544         print("Compilation failed")
545         exit()

```

Figure 9

boolFactor rule:

• Initializing Variables:

- Creates the lists `Rtrue` and `Rfalse` blanks.

• Controlling the Press of the First tokens:

- If the first token is a command (item `commandtoken`), then it must be an expression or a condition.
 - o If it is an expression, it checks for the comparison operator and a second expression.
 - o If it is condition with `then` or `not`, parses the corresponding condition and returns the opposite sets `Rtrue` and `Rfalse`.
- If the first token is not a command, then it can be either an expression or a condition.


```

547 def isBoolFactor(token):
548     Rfalse = []
549     if token[0] == "commandtoken":
550         expr1 = isExpression(token)
551         if expr1[0]:
552             E1place = expr1[1]
553             token = lex()
554             if token[0] in relOpList:
555                 if token[0] == "isothtatoke":
556                     relOp = "="
557                 elif token[0] == "mikroterotoken":
558                     relOp = "<"
559                 elif token[0] == "megaluterotoken":
560                     relOp = ">"
561                 elif token[0] == "mikisotoken":
562                     relOp = "<="
563                 elif token[0] == "megisotoken":
564                     relOp = ">="
565                 else:
566                     relOp = "!="
567             token = lex()
568             expr2 = isExpression(token)
569             if expr2[0]:
570                 E2place = expr2[1]
571                 Rtrue = makeList(nextQuad())
572                 quadList.append(genQuad(relOp, E1place, E2place, "_"))
573                 Rfalse = makeList(nextQuad())
574                 quadList.append(genQuad("jump", "_", "_", "_"))
575                 return [True, Rtrue, Rfalse]
576             else:
577                 print("At line", token[1], "an expression was expected after a relationship operand")
578                 print("Compilation failed")
579                 exit()
580

```

Figure 10

- **Apply Comparison Operator:**

- If the next token is a comparison operator, then it performs the corresponding comparison with the next expression.

- **Return Results:**

- Returns a list of three values: [True, Rtrue, Rfalse] if the analysis is successful, otherwise returns [False].

```

581         elif token[2] == "not":
582             token = lex()
583             cond = isCondition(token)
584             if cond[0]:
585                 Btrue = cond[1]
586                 Bfalse = cond[2]
587                 Rtrue = Bfalse
588                 Rfalse = Btrue
589                 return [True,Rtrue,Rfalse]
590             else:
591                 print("At line:", token[1],"a boolean factor was expected after 'not'")
592                 print("Compilation failed")
593                 exit()
594         else:
595             return [False]
596     else:
597         expr1 = isExpression(token)
598         if expr1[0]:
599             E1place = expr1[1]
600             token = lex()
601             if token[0] in relOpList:
602                 if token[0] == "isothtatoke":
603                     relOp = "="
604                 elif token[0] == "mikroterotoken":
605                     relOp = "<"
606                 elif token[0] == "megaluterotoken":
607                     relOp = ">"
608                 elif token[0] == "mikisotoken":
609                     relOp = "<="
610                 elif token[0] == "megisotoken":
611                     relOp = ">="
612                 else:
613                     relOp = "!="
614             token = lex()
615             expr2 = isExpression(token)
616             if expr2[0]:
617                 E2place = expr2[1]
618                 Rtrue = makeList(nextQuad())
619                 quadList.append(genQuad(relOp, E1place,E2place,"_"))
620                 Rfalse = makeList(nextQuad())
621                 quadList.append(genQuad("jump", "_", "_", "_"))
622                 return [True,Rtrue,Rfalse]

```

Figure 11

```

621         quadList.append(genQuad("jump", "_", "_", "_"))
622         return [True,Rtrue,Rfalse]
623     else:
624         print("At line", token[1], "an expression was expected after a relationship operand")
625         print("Compilation failed")
626         exit()
627
628     else:
629         cond = isCondition(token)
630         if cond[0]:
631             Btrue = cond[1]
632             Bfalse = cond[2]
633             Rtrue = Btrue
634             Rfalse = Bfalse
635             return [True,Rtrue,Rfalse]
636         else:
637             return [False]

```

Figure 12

Rule expression:

- **Initializing Variables:**

- Initially, variables such as `Eplace`, `seekIndex`, `negSign`.

- **Optional Premise Check:**

- If the current token has an optional token (`addtoken` or `subtoken`), it parses the next token.
- If the token is a `subtoken`, then the variable `negSign` defined as `True`.

```
638 def isExpression(token):
639     Eplace = 0
640     global seekIndex
641     global pos
642     global scope
643     seekIndex = f.tell()
644     negSign = False
645     if isOptionalSign(token):
646         if token[0] == "subtoken":
647             negSign = True
648             token = lex()
649     term1 = isTerm(token)
650     if term1[0]:
651         T1place = term1[1]
652         if negSign:
653             y = newTemp()
654             pos=pos+4
655             newPos=pos
656             newScope=scope-1
657             recordStructure.addNewEntity(scopeIndex=newScope,entityName=y,entity_type='TemporaryVariable', offset=newPos)
658             quadList.append(genQuad("-",0,T1place,y))
659             T1place = y
660             seekIndex = f.tell()
661             token = lex()
```

Figure 13

- **Analysis of Terms:**

- The numerical terms of the expression are analyzed.
- If an optional "-" sign is present, reverse orientation is applied.
- For each `addtoken` or `subtoken` that follows, a new term is parsed and the appropriate value assignment is made.

- **Return Results:**

- Returns `True` and the place of the result (`Eplace`) if the analysis is successful.
- Otherwise, it returns `False`.

```

662 while token[0] == "addtoken" or token[0] == "subtoken":
663     if token[0] == "addtoken":
664         operand = "+"
665     else:
666         operand = "-"
667     token = lex()
668     term2 = isTerm(token)
669     if term2[0]:
670         T2place = term2[1]
671         w = newTemp()
672         pos=pos+4
673         newPos=pos
674         newScope=scope-1
675         recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
676         quadList.append(genQuad(operand,T1place,T2place,w))
677         T1place = w
678         seekIndex = f.tell()
679         token = lex()
680     else:
681         print("At line", token[1], "a term was expected after an add or sub opperand. Instead found:", token[0])
682         print("Compilation failed")
683         exit()
684     f.seek(seekIndex)
685     Eplace = T1place
686     return [True,Eplace]
687 else:
688     f.seek(seekIndex)
689     return [False]

```

Figure 14

optionalSign rule:

Checking the token:

- If the current token is addtoken or subtoken, then it is considered an optional token and the function returns True.
- In any other case, it returns False.

```

691 def isOptionalSign(token):
692     global seekIndex
693     if token[0] == "addtoken" or token[0] == "subtoken":
694         return True
695     else:
696         return False

```

Figure 15

Term rule:

- **Initializing Variables:**

- Variables such as n are initialized Tplace, seekIndex.

- **Factor Analysis:**

- The numerical factors of the term are analyzed.
- For each subsequent multiplication or division, a new factor is analyzed and assigned the appropriate values.

- **Return Results:**

- Returns True and the position of the result (Tplace) if the analysis is successful.
- Otherwise, it returns False.

```
698 def isTerm(token):
699     Tplace = 0
700     global seekIndex
701     global pos
702     global scope
703     factor1 = isFactor(token)
704     if factor1[0]:
705         F1place = factor1[1]
706         seekIndex = f.tell()
707         token = lex()
708         while token[0] == "multoken" or token[0] == "modtoken" or token[0] == "divtoken":
709             if token[0] == "multoken":
710                 operand = "*"
711             elif token[0] == "divtoken":
712                 operand = "/"
713             else:
714                 operand = "mod"
715             token = lex()
716             factor2 = isFactor(token)
717             if factor2[0]:
718                 F2place = factor2[1]
719                 w = newTemp()
720                 pos=pos+4
721                 newPos=pos
722                 newScope=scope-1
723                 recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
724                 quadList.append(genQuad(operand,F1place,F2place,w))
725                 F1place = w
726                 seekIndex = f.tell()
727                 token = lex()
728             else:
729                 print("At line", token[1], "a factor was expected after a multiplication or division operand. Instead found: ",token[0])
730                 print("Compilation failed")
731                 exit()
732         f.seek(seekIndex)
733         Tplace = F1place
734         return [True,Tplace]
735     else:
736         return [False]
```

Figure 16

factor rule:

- **Controltokens:**

- If the current token is a numbertoken, the function returns its value and True.

- If the token is an identifier (anagoristikotoken), it is checked if the token's tail follows.
 - If the token is a left parenthesis (leftpartoken), the following expression is analyzed and the corresponding right parenthesis is expected.
- **Return Results:**
- Returns True and the factor's position (Fplace) if the analysis is successful.
 - Otherwise, it returns False.

```

738 def isFactor(token):
739     Fplace = 0
740     global seekIndex
741     global pos
742     global scope
743     if token[0] == "numbertoken":
744         Fplace = token[2]
745         return [True, Fplace]
746     elif token[0] == "anagoristikotoken":
747         idName = token[2]
748         seekIndex = f.tell()
749         tempSeekIndex = seekIndex
750         token = lex()
751         if idTail(token):
752             w = newTemp()
753             pos=pos+4
754             newPos=pos
755             newScope=scope-1
756             recordStructure.addNewEntity(scopeIndex=newScope,entityName=w,entity_type='TemporaryVariable', offset=newPos)
757             quadList.append(genQuad("par",w,"ret"," "))
758             quadList.append(genQuad("call",idName,"_","_"))
759             return [True,w]
760         else:
761             IDplace = idName
762             Fplace = IDplace
763             f.seek(tempSeekIndex)
764             return [True, Fplace]
765     elif token[0] == "leftpartoken":
766         token = lex()
767         expr = isExpression(token)
768         if expr[0]:
769             Eplace = expr[1]
770             Fplace = Eplace
771             token = lex()
772             if token[0] == "rightpartoken":
773                 return [True, Fplace]
774             else:
775                 print("At line", token[1], "a ')' was expected")
776                 print("Compilation failed")
777                 exit()
778         else:
779             print("At line", token[1], "an expression was expected")
780             print("Compilation failed")
781             exit()
782     else:
783         return [False]

```

Figure 17

idTail rule:

- **ID Check:** If the current token is a left parenthesis (leftpartoken), then checks if a list of parameters (parList) follows and then expects a right parenthesis (rightpartoken).

- **Return Result:** Returns True if the queue is syntactically correct, that is, if there are left and right parentheses after a list of parameters. Otherwise, it returns False.

```
785     def idTail(token):
786         if token[0] == "leftpartoken":
787             token = lex()
788             if parList(token):
789                 token = lex()
790                 if token[0] == "rightpartoken":
791                     return True
792                 else:
793                     print("At line", token[1], "a ')' was expected")
794                     print("Compilation failed")
795                     exit()
796             else:
797                 return False
798         else:
799             return False
```

Figure 18

parList rule:

- **Expression Analysis:** First checks if an expression exists. If so, then it stores the result of the expression in a variable and adds the parameter to the stack with the appropriate type (cvfor the guest Department). It also notes that at least one parameter exists.
- **Repetition:** If there is a parameter, the function repeats the process for each subsequent parameter that follows the separator,.
- **Return Result:** Returns True if there are any parameters in the list or False if there are no parameters.

```

801 def parList(token):
802     global seekIndex
803     global isAtLeastOnePar
804     global pos
805     global scope
806     expr1 = isExpression(token)
807     if expr1[0]:
808         E1place = expr1[1]
809         a = E1place
810         pos=pos+4
811         newPos=pos
812         newScope=scope-1
813         recordStructure.addNewEntity(scopeIndex=newScope,entityName=a,entity_type='Variable', offset=newPos)
814         quadList.append(genQuad("par",a,"CV","_"))
815         isAtLeastOnePar = True
816         seekIndex = f.tell()
817         token=lex()
818         while token[0] == "commatoken":
819             seekIndex = f.tell()
820             token = lex()
821             expr2 = isExpression(token)
822             if expr2[0]:
823                 E2place = expr2[1]
824                 b = E2place
825                 pos=pos+4
826                 newPos=pos
827                 newScope=scope-1
828                 recordStructure.addNewEntity(scopeIndex=newScope,entityName=b,entity_type='Variable', offset=newPos)
829                 quadList.append(genQuad("par",b,"CV","_"))
830                 seekIndex = f.tell()
831                 token=lex()
832                 continue
833             else:
834                 print("At line", token[1], "an expression was expected after the ','")
835                 print("Compilation failed")
836                 exit()
837         f.seek(seekIndex)
838         return isAtLeastOnePar
839     else:
840         f.seek(seekIndex)
841         return False

```

Figure 19

Statement rule:

1. First it checks if the statement is a simple statement (simple statement) with the using the function `isSimpleStatement(token)`.
2. If the statement is not simple, it checks if it is a structured statement using the function `isStructuredStatement(token)`.

If control in either case returns `True`, then the function also returns `True`, indicating that the input is a statement. Otherwise, it returns `False`.

```

843 def isStatement(token):
844     if isSimpleStatement(token):
845         return True
846     elif isStructuredStatement(token):
847         return True
848     else:
849         return False
850

```

Figure 20

simple_statement rule:

1. It first checks if the statement is an assignment using the function `isAssignmentStat(token)`.
2. If the statement is not an assignment, it checks if it is a print statement using the function `isPrintStat(token)`.
3. If the statement is neither an assignment nor a print statement, it checks whether it is a return statement using the function `isReturnStat(token)`.

If the control in any of the above cases returns `True`, then the function also returns `True`, indicating that the input is a simple statement. Otherwise, it returns `False`.

```
851     def isSimpleStatement(token):
852         if isAssignmentStat(token):
853             return True
854         elif isPrintStat(token):
855             return True
856         elif isReturnStat(token):
857             return True
858         else:
859             return False
```

Figure 21

assignment_stat rule:

- Check for ID (identifier): It first checks if the statement starts with an identifier, which is recognized by the "anagnotistikotoken" token. If not, the statement is not an assignment and the function returns `False`.
- Check for the assignment operator: If the first token is an identifier, checks if the assignment operator "=" follows the token "ana8eshtoken". If not, the statement is not an assignment and the function returns `False`.
- Check for expression: If the previous checks pass, the function checks if an expression follows the assignment operator. This is done by calling the function `isExpression(token)`. If a valid expression exists, a quadruple is created for the assignment using the function `genQuad(":=", Eplace, "_", idPlace)` and the function returns `True`. If no valid expression, an error message is displayed and the function returns `False`.

```

861 def isAssignmentStat(token):
862     if token[0] == "anagnotistikotoken":
863         idPlace = token[2]
864         token = lex()
865         if token[0] == "ana8eshtoken":
866             token = lex()
867             expr = isExpression(token)
868             if expr[0]:
869                 Eplace = expr[1]
870                 quadList.append(genQuad(":", Eplace, "_", idPlace))
871                 return True
872             elif token[0] == "commandtoken":
873                 if token[2] == "int":
874                     token = lex()
875                     if token[0] == "leftpartoken":
876                         token = lex()
877                         if token[0] == "commandtoken":
878                             if token[2] == "input":
879                                 token = lex()
880                                 if token[0] == "leftpartoken":
881                                     token = lex()
882                                     if token[0] == "rightpartoken":
883                                         token = lex()
884                                         if token[0] == "rightpartoken":
885                                             quadList.append(genQuad("inp", idPlace, "_", "_"))
886                                             return True
887                                         else:
888                                             print("At line", token[1], "a ')' was expected")
889                                             print("Compilation failed")
890                                             exit()
891                                     else:
892                                         print("At line", token[1], "a '(' was expected")
893                                         print("Compilation failed")
894                                         exit()
895                                 else:
896                                     print("At line", token[1], "a '(' was expected")
897                                     print("Compilation failed")
898                                     exit()
899                             else:
900                                 return False
901                         else:
902                             return False
903                     else:
904                         print("At line", token[1], "a '(' was expected")
905                         print("Compilation failed")
906                         exit()
907                 else:
908                     return False

```

Figure 22

• Additional Check for 'int(input())': If the expression is false and the next token is the statement "int", the function performs an additional check to confirm that the statement is of the form "int(input())". If this is true, a triple is created for the input using the function `genQuad("inp", idPlace, "_", "_")` and the function returns True. If not is of the expected form

```

909     else:
910         print("At line", token[1], "either an expression or 'int(input())' was expected after the '='")
911         print("Compilation failed")
912         exit()
913     else:
914         return False
915     else:
916         return False

```

Figure 23

return_stat rule:

- Check for return command: First checks if the firsttoken is the "return" command, which is identified by the "commandtoken" token. If it is not, the function returnsFalse.
- Check for expression: If the firsttoken is the "return" command, then it checks if an expression follows. This is done by calling the function isExpression(token). If a valid expression exists, a quadruple is created to return using the functiongenQuad("ret", Eplace, "_", "_")and the function returnsTrue. If there is no valid expression, an error message is displayed and the function returnsFalse.

```
918 def isReturnStat(token):
919     if token[0] == "commandtoken":
920         if token[2] == "return":
921             token = lex()
922             expr = isExpression(token)
923             if expr[0]:
924                 Eplace = expr[1]
925                 quadList.append(genQuad("ret", Eplace, "_", "_"))
926                 return True
927             else:
928                 print("At line", token[1], "an expression was expected after the 'return' command")
929                 print("Compilation failed")
930                 exit()
931         else:
932             return False
933     else:
934         return False
```

Figure 24

structure_statement rule:

- Check for statementif: Calls the functionifState(token)to check if token represents an if statement. If so, it returnsTrue.
- Check for statementwhile: Calls the functionwhileState(token)to check if the token represents a while statement. If so, it returnsTrue.
- ReturnFalse: If the token does not correspond to either an if statement or a while statement, it returnsFalse.

```
936 def isStructuredStatement(token):
937
938     if ifState(token):
939         return True
940     elif whileState(token):
941         return True
942     else:
943         return False
```

Figure 25

while_stat rule:

- Check if the token represents the statement while: It first checks if the token is a valid command (command token) and if it matches the "while" keyword. If not, it returns False.
- Analysis of the body while loop: After checking that the statement begins with the "while" keyword, it continues checking in the following order:
 - It checks the condition of the while loop using the function isCondition(token).
 - If the condition is valid, then it checks if it is followed by the correct '{' symbol with the function lex().
 - Next is checking for the statements inside the body of the while loop using the function statements(token).
 - Checks whether the body of the while loop is closed with the appropriate '}' symbol.
 - If all of the above is valid, it returns True.
- Return False: If the token does not represent a while statement, then it returns False.

```
945 def whileState(token):
946     if token[0] == "commandtoken":
947         if token[2] == "while":
948             condQuad = nextQuad()
949             token = lex()
950             cond = isCondition(token)
951             if cond[0]:
952                 conditionTrue = cond[1]
953                 conditionFalse = cond[2]
954                 backpatch(conditionTrue, nextQuad())
955                 token = lex()
956                 if token[0] == "anwkatwtoken":
957                     token = lex()
958                     if token[0] == "anoigmatoken":
959                         token = lex()
960                         if statements(token):
961                             token = lex()
962                             if token[0] == "kleisimotoken":
963                                 quadList.append(genQuad("jump", " ", " ", condQuad))
964                                 backpatch(conditionFalse, nextQuad())
965                                 return True
966                             else:
967                                 print("At line", token[1], "a '#' was expected to close a '#{ found earlier for the while loop}")
968                                 print("Compilation failed")
969                                 exit()
970                             else:
971                                 print("At line", token[1], "a statement was expected after the ':'")
972                                 print("Compilation failed")
973                                 exit()
974                             else:
975                                 print("At line", token[1], "a '#' was expected after the 'while'")
976                                 print("Compilation failed")
977                                 exit()
978                             else:
979                                 print("At line", token[1], "a ':' was expected")
980                                 print("Compilation failed")
981                                 exit()
982                             else:
983                                 print("At line", token[1], "a condition was expected after the 'while' command")
984                                 print("Compilation failed")
985                                 exit()
986                             else:
987                                 return False
988                         else:
989                             return False
990                     else:
991                         return False
```

Figure 26

startRule rule:

- **Download nexttokens:** First, it calls the function `lex()` to receive the first program token.
- **Create a new scope level:** Adds a new level of scope to the record structure for variables and functions.
- **Check statements:** Checks if there are declarations in the program using the function `declarations(token)`. If the declaration process is successful, it continues, otherwise it terminates the compilation.
- **Checking and calling functions:** Checks and calls the function `isDefFunctions(token)` for each subsequent token until there are no more function declarations in the program.
- **Checking and calling the main section (main):** Checks if the main part of the program with the function exists `callMainPart(token)`. If all are valid, it adds the corresponding trailing quaternions (`halt`, `end_block`), displays the final quad and returns `True`. If the main section is not found, the program terminates with an error.

```
992     def startRule():
993         token=lex()
994         recordStructure.addNewScope()
995         if declarations(token):
996             token=lex()
997             while isDefFunctions(token):
998                 token=lex()
999             if callMainPart(token):
1000                 quadList.append(genQuad("halt","_","_","_"))
1001                 quadList.append(genQuad("end_block","main","_","_"))
1002                 readQuadList(quadList[-1])
1003                 return True
1004             else:
1005                 exit()
1006         else:
1007             print("At line", token[1]," declarations was expected")
1008             print("Compilation failed")
1009             exit()
```

Figure 27

print_stat rule:

1. **Command type check:** First, it checks if the current token is a command. If it isn't, it returns `False`.

2. **Print order check:** If the current token is a print command (print), continues checking, otherwise returns False.
3. **Left bracket control:** Waits for the next token to be the left bracket (.). If it is not, it displays an error message and terminates the compilation.
4. **Print expression control:** Expects an expression after the left bracket. If an expression exists, it continues checking, otherwise it displays an error message and terminates compilation.
5. **Right bracket control:** Expects the right bracket ()) after the expression. If this is not the case, it displays an error message and terminates the compilation.

If all steps pass successfully, it adds a block of appropriate print command (out) in the final list of quads and returns True.

```

1011 def isPrintStat(token):
1012     global seekIndex
1013     if token[0] == "commandtoken":
1014         if token[2] == "print":
1015             token = lex()
1016             if token[0] == "leftpartoken":
1017                 token = lex()
1018                 expr = isExpression(token)
1019                 if expr[0]:
1020                     Eplace = expr[1]
1021                     token = lex()
1022                     if token[0] == "rightpartoken":
1023                         quadList.append(genQuad("out", Eplace, "_", "_"))
1024                         seekIndex = f.tell()
1025                         return True
1026                     else:
1027                         print("At line", token[1], "a ')' was expected")
1028                         print("Compilation failed")
1029                         exit()
1030                 else:
1031                     print("At line", token[1], "an expression was expected after the 'print' command")
1032                     print("Compilation failed")
1033                     exit()
1034             else:
1035                 print("At line", token[1], "a '(' was expected after the 'print' command")
1036                 print("Compilation failed")
1037                 exit()
1038         else:
1039             return False
1040     else:
1041         return False

```

Figure 28

id_List rule:

- **Initialize variables:** Initially, two variables are defined, hseekIndex and theisAtLeastOneID.
- **ID check:** If the first token is an identifier, it is assumed that there is at least one identifier in the list and its position is recorded in the variable seekIndex. The identifier is added to the listnames.
- **Repeat for more IDs:** If a comma follows, checks if another identifier follows. If so, it adds it to the listnamesand

records its position with `seekIndex`. It repeats this process until the list of IDs is finished.

- **Return results:** Returns a list containing a logical boolean (`isAtLeastOneID`) which indicates if there is at least one identifier and a list (`names`) with the identifiers found.

```

0  def isIdList(token):
1      global seekIndex
2      global isAtLeastOneID
3      global pos
4      global scope
5      names = []
6      if token[0]=="anagnoristikotoken":
7          isAtLeastOneID = True
8          seekIndex = f.tell()
9          names.append(token[2])
10         token=lex()
11         newScope=scope-1
12         while token[0]=="commatoken":
13             token=lex()
14             if token[0]=="anagnoristikotoken":
15                 names.append(token[2])
16                 seekIndex = f.tell()
17                 token=lex()
18             else:
19                 print("At line", token[1]," an id was expected")
20                 print("Compilation failed")
21                 exit()
22         f.seek(seekIndex)
23         for x in names:
24             pos=pos+4
25             newPos=pos
26             recordStructure.addNewEntity(scopeIndex=newScope,entityName=x,entity_type='Variable', offset=newPos)
27         return [isAtLeastOneID,names]
28     else:
29         f.seek(seekIndex)
30         return [False,names]

```

Figure 29

Rule statements:

- **Check statement:** First, it checks if the first token represents a statement. If not, it returns `False`.
- **Repeat for more statements:** If the first token represents a statement, then it iterates to check if there are more statements in the sequence. If not, it returns `True`.
- **Return result:** Returns `True` if all statements are valid, otherwise `False`.

```

1068     def statements(token):
1069         global seekIndex
1070         if isStatement(token) == False:
1071             f.seek(seekIndex)
1072             return False
1073         f.seek(seekIndex)
1074         token = lex()
1075         while isStatement(token):
1076             seekIndex = f.tell()
1077             f.seek(seekIndex)
1078             token=lex()
1079
1080         f.seek(seekIndex)
1081         return True
1082

```

Figure 30

Rule declarations:

- **Check statement:** First, it checks if the first token represents a statement. If there is at least one statement, the variable `isAtLeastOneDeclaration` is set to `True`.
- **Repeat for more statements:** If there is at least one statement and the variable `isAtLeastOneDeclaration` is `True`, then an iteration is performed to test the remaining statements as well.
- **Return result:** Returns the value of the variable `isAtLeastOneDeclaration`, which indicates whether there is at least one valid statement or not.

```

1083     def declarations(token):
1084         global seekIndex
1085         global isAtLeastOneDeclaration
1086         seekIndex = f.tell()
1087         isAtLeastOneDeclaration = isDeclaration(token)
1088         token=lex()
1089         while isDeclaration(token) and isAtLeastOneDeclaration:
1090             seekIndex = f.tell()
1091             token = lex()
1092         f.seek(seekIndex)
1093         return isAtLeastOneDeclaration
1094

```

Figure 31

Rule globals:

- **Check statement:** First, it checks if the first token represents a declaration for a global variable. If there is at least one statement, the variable `isAtLeastOneGlobal` is set to `True`.
- **Repeat for more statements:** If there is at least one statement and the variable `isAtLeastOneGlobal` is `True`, then an iteration is performed to check the remaining declarations for global variables as well.
- **Return result:** Returns the value of the variable `isAtLeastOneGlobal`, which indicates whether there is at least one valid declaration for a global variable or not.

```
1095     def globalVar(token):
1096         global seekIndex
1097         global isAtLeastOneGlobal
1098         seekIndex = f.tell()
1099         isAtLeastOneGlobal = isGlobal(token)
1100         token=lex()
1101         while isGlobal(token):
1102             seekIndex = f.tell()
1103             token=lex()
1104         f.seek(seekIndex)
1105         return isAtLeastOneGlobal
```

Figure 32

Declaration rule:

- **Statement type check:** First, it checks if the first token is one `intdef` token, which indicates a type variable declaration `int`.
- **Analysis of `itidList`:** If the declaration is of type `int`, continues checking with the function `isIdList` to check if it is followed by a valid list identifiers.

- **Return result:** Returns `True` if the statement is valid, that is if it starts with `intdef` token and followed by a valid list of identifiers, otherwise it returns `False`.

```

1107     def isDeclaration(token):
1108         if token[0]=="intdef":
1109             token=lex()
1110             idList =isIdList(token)
1111             if idList[0]:
1112                 return True
1113             else:
1114                 print("At line", token[1], "an id was expected ")
1115                 print("Compilation failed")
1116                 exit()
1117         else:
1118             return False
1119

```

Figure 33

Global rule:

- **Check declaration type:** First, it checks if the first token is one `command` token and if the command is `global`.
- **Analysis of `idList`:** If the command is of type `global`, continues checking with the function `isIdList` to check if it is followed by a valid list identifiers.
- **Return result:** Returns `True` if the command is valid, that is if it starts with `global` and followed by a valid list of identifiers, otherwise returns `False`.

```

1120     def isGlobal(token):
1121         if token[0]=="command":
1122             if token[2]=="global":
1123                 token=lex()
1124                 idList =isIdList(token)
1125                 if idList[0]:
1126                     return True
1127                 else:
1128                     print("At line", token[1], "an id was expected ")
1129                     print("Compilation failed")
1130                     exit()
1131             else:
1132                 return False
1133         else:
1134             return False

```

Figure 34

def_function rule:

- **Check the first onetokens:**

- It starts by checking if the first token is onecommandtoken and if the command is def. This indicates that a function declaration is found.

- **Parsing the function name:**

- If the current token is isdef, then we expect the function name to follow (aanagnotistikotoken). This name is stored in the variable q.
- Then we wait for oneleftpartoken, indicating the beginning of the function's argument list.
- The following is the parsing of the function's argument list, which is done by calling the function isIdList.
- Then we store the length of the function frame (which is calculated based on the number of arguments) in the variable framelen.
- Additionally, we create a new record in the program's data structure to store the function, including its arguments.

```
1136 def isDefFunctions(token):
1137     global scope
1138     global pos
1139     if token[0] == "commandtoken":
1140         if token[2] == "def":
1141             token = lex()
1142             if token[0] == "aanagnotistikotoken":
1143                 lastBlock.append(token[2])
1144                 q = token[2]
1145                 pos = pos + 4
1146                 newPos = pos
1147                 newScope = scope + 1
1148                 token = lex()
1149                 if token[0] == "leftpartoken":
1150                     token = lex()
1151                     idList = isIdList(token)
1152                     if idList[0]:
1153                         framelen = newPos
1154                         recordStructure.addNewEntity(scopeIndex = newScope, entityName = q, entity_type = 'Function', frameLength = newPos, arguments = idList[1])
1155                         token = lex()
1156                         if token[0] == "rightpartoken":
1157                             token = lex()
1158                             if token[0] == "anwkatwtoken":
1159                                 recordStructure.addNewScope()
1160                                 token = lex()
1161                                 if token[0] == "anoigmatoken":
1162                                     token = lex()
1163                                     if declarations(token):
1164                                         token = lex()
1165                                         hasFunction = False
1166                                         while True:
1167                                             if isDefFunctions(token):
1168                                                 hasFunction = True
1169                                                 token = lex()
1170                                                 continue
1171                                         else:
1172                                             if hasFunction == False:
1173                                                 quadList.append(emptyList())
1174                                                 indexList.append(len(quadList) - 1)
1175                                             break
```

Figure 35

```

1194         if statements(token):
1195             token = lex()
1196
1197         if token[0] == "kleisimotoken":
1198             quadList.append(genQuad("end_block",lastBlock[-1],"_","_"))
1199
1200             pos=newPos
1201             if len(lastBlock) == 1 and len(indexList) == 0:
1202                 lastBlock.pop()
1203                 readQuadList(quadList[-1])
1204                 return True
1205             if len(indexList)>0:
1206                 quadList[indexList[-1]] = ["begin_block",lastBlock.pop(),"_","_"]
1207
1208                 indexList.pop()
1209                 readQuadList(quadList[-1])
1210                 if len(indexList) == 0 and len(lastBlock)>0:
1211                     quadList.append(genQuad("begin_block",lastBlock[-1],"_","_"))
1212             else:
1213                 quadList.append(genQuad("begin_block",lastBlock[-1],"_","_"))
1214
1215                 return True
1216             else:
1217                 print ("At line", token[1], "a '#' was expected to close a '#{ ' found earlier")
1218                 print("Compilation failed")
1219                 exit()
1220             else:
1221                 print ("At line", token[1], "a statement was expected")
1222                 print("Compilation failed")
1223                 exit()
1224             else:
1225                 print ("At line", token[1], "a '#{ ' was expected")
1226                 print("Compilation failed")
1227                 exit()
1228             else:
1229                 print ("At line", token[1], "a ':' was expected")
1230                 print("Compilation failed")
1231                 exit()
1232             else:
1233                 print ("At line", token[1], "a ')' was expected")
1234                 print("Compilation failed")
1235                 exit()
1236             else:
1237                 print ("At line", token[1], "a ' ' was expected")
1238                 print("Compilation failed")
1239                 exit()

```

Figure 36

- **Analysis of the body of the function:**

- Following is the analysis of the function body.
- Statements are parsed using the function declarations.
- Check for other built-in functions using it isDefFunctions.
- The following is the analysis of the variables (global variables) using it globalVar.
- Finally, the analysis of the commands of the body of the function follows use of itstatements.

- **Completion of the analysis:**

- Completes parsing of the function body.
- Adds the necessary final notebooks (genQuad) for the output.

```

1218         else:
1219             print ("At line", token[1], "a ')' was expected")
1220             print("Compilation failed")
1221             exit()
1222         else:
1223             print ("At line", token[1], "an id was expected")
1224             print("Compilation failed")
1225             exit()
1226         else:
1227             print ("At line", token[1], "a '(' was expected")
1228             print("Compilation failed")
1229             exit()
1230         else:
1231             print ("At line", token[1], "an id was expected")
1232             print("Compilation failed")
1233             exit()
1234         else:
1235             return False
1236     else:
1237         return False
1238

```

Figure 37

- **Addition of the final quatrains:**

- Endnotes are added to terminate the function.
- If there is only one function layer and no built-in functions (isDefFunctions), then the command is takenhaltfor termination of the program.
- The final notebook is enteredend_blockto mark its end body of the function.

- **Return result:**

- ReturnsTrueafter the function analysis has completed successfully.
- If any of the expected commands are missing, the program prints an error message and terminates.

call_main_part rule:

- **Check function definitionmain:** First, it checks if the first token is defitokenand whether the function has a namemain.

- **Create function record:** If the functionmainit is valid, adds a record for the functionmainin the data structure recordStructure. This record includes the name, framelength, which in your case is the size of the main data structurescope(framelen), as well as the argument list (empty for the main).

- **Creation of the top four:** Adds a special quad that indicates the start of the main code section.

- **Check statements and orders:** Checks the statements and statements that follow the function definitionmain.
- **Return result:** ReturnsTrueif the function definitionmain is valid and valid statements and commands follow, otherwise returns False.

```

1240 def callMainPart(token):
1241     global scope
1242     global pos
1243     global framelen
1244     recordStructure.addNewScope()
1245     if token[0] == "defitoken":
1246         token = lex()
1247         if token[0] == "commandtoken":
1248             if token[2] == "main":
1249                 q = token[2]
1250                 pos=pos+4
1251                 newPos=pos
1252                 newScope=scope+1
1253                 framelen=newScope
1254                 recordStructure.addNewEntity(scopeIndex=newScope,entityName=q,entity_type='Function', framelength=newPos,arguments=[])
1255                 quadList.append(genQuad("begin_block","main"," "," "))
1256                 token=lex()
1257                 if declarations(token):
1258                     token=lex()
1259                 if statements(token):
1260                     return True
1261             else:
1262                 print ("At line", token[1], "a statement was expected")
1263                 print("Compilation failed")
1264                 exit()
1265         else:
1266             exit()
1267     else:
1268         exit()
1269     else:
1270         return False

```

Figure 38

5) Intermediate code

To implement the intermediate code, the auxiliary functions (their implementation is shown in Figure 6) described in the slides and the handbook of the course were used and placed in the appropriate places with the appropriate parameters (according to the handbook), in order to produce the intermediate code . The description of the above can be seen in chapter 4 in detail.

```

1297     def nextQuad():
1298         global quadNum
1299         return quadNum+1
1300
1301     def genQuad(op,x,y,z):
1302         global quadNum
1303         quadNum += 1
1304         quad = [op,x,y,z]
1305         return quad
1306
1307     def newTemp():
1308         global tempNum
1309         tempNum += 1
1310         s = "T_"
1311         return s+str(tempNum)
1312
1313     def emptyList():
1314         return genQuad("_","_","_","_")
1315
1316     def makeList(label):
1317         return [label]
1318
1319     def mergeList(list1,list2):
1320         return list1+list2
1321
1322     def backpatch(lst, label):
1323         for x in lst:
1324             quadList[x-1][-1] = label
1325

```

Figure 39

6)Table of symbols

For the symbol table the RecordStructure class was created (Figures 40,41&42) which implements the structure (according to the slides and the handbook) for the symbol table. By using the methods of this structure in the appropriate places the table is created. More specifically, the global variable scope is used which defines the scope of the table and pos which defines the offset. pos is incremented by 4 when an entity is created. The scope increases by 1 when a function is created and decreases by 1 when it reaches its end. The description of the above can be seen in chapter 4 in detail.

```

139
140 + class RecordStructure:
141     def __init__(self):
142         self.scopes = []
143
144     class RecordScope:
145         def __init__(self, entityList, nestingLevel):
146             self.entityList = entityList
147             self.nestingLevel = nestingLevel
148
149     class RecordEntity:
150         def __init__(self, name):
151             self.name = name
152
153     class Variable(RecordEntity):
154         def __init__(self, name, offset):
155             super().__init__(name)
156             self.offset = offset
157
158     class Function(RecordEntity):
159         def __init__(self, name, framelength, arguments):
160             super().__init__(name)
161             self.framelength = framelength
162             self.arguments = arguments
163
164     class TemporaryVariable(RecordEntity):
165         def __init__(self, name, offset):
166             super().__init__(name)
167             self.offset = offset
168
169     class RecordArgument:
170         def __init__(self, parMode):
171             self.parMode = parMode
172

```

Figure 40

Internal Classes

1. RecordScope:

- o Represents a scope with a list of entities (entityList) and a level of nesting (nestingLevel).
- o `__init__(self, entityList, nestingLevel)`: Manufacturer which initializes the entity list and nesting level.

2. RecordEntity:

- o The base class for all entities, contains only the name of the entity.
- o `__init__(self, name)`: Constructor that initializes the name.

3. Variable (inherits from RecordEntity):

- o Represents a variable with a name and an offset (offset).
 - o `__init__(self, name, offset)`: Constructor that initializes name and offset.
- 4.Function**(inherits from `RecordEntity`):
- o Represents a function with a name, frame length (framelength) and argument list (arguments).
 - o `__init__(self, name, framelength, arguments)`: Constructor that initializes name, frame length, and arguments.
- 5.TemporaryVariable**(inherits from `RecordEntity`):
- o Represents a temporary variable with a name and an offset (offset).
 - o `__init__(self, name, offset)`: Constructor that initializes name and offset.
- 6.RecordArgument**:
- o Represents a function argument with a parameter function (parMode).
 - o `__init__(self, parMode)`: Constructor that initializes the parameter function.

```

174 def addNewScope(self):
175     newScope = RecordStructure.RecordScope(entityList=[], nestingLevel=len(self.scopes) + 1)
176     self.scopes.append(newScope)
177     return newScope
178
179 def scopeDeletion(self, scopeIndex):
180     if scopeIndex < len(self.scopes):
181         self.scopes.pop(scopeIndex)
182
183
184 def addNewEntity(self, scopeIndex, entityName, entity_type, **kwargs):
185     if scopeIndex < len(self.scopes):
186         if entity_type == 'Variable':
187             newEntity = RecordStructure.Variable(name=entityName, **kwargs)
188         elif entity_type == 'Function':
189             newEntity = RecordStructure.Function(name=entityName, **kwargs)
190         elif entity_type == 'TemporaryVariable':
191             newEntity = RecordStructure.TemporaryVariable(name=entityName, **kwargs)
192         else:
193             return None
194         self.scopes[scopeIndex].entityList.append(newEntity)
195         return newEntity
196
197 def addNewArgument(self, scopeIndex, entityIndex, argumentName, parMode, type):
198     if scopeIndex < len(self.scopes) and entityIndex < len(self.scopes[scopeIndex].entityList):
199         newArgument = RecordStructure.RecordArgument(parMode=parMode, type=type)
200         if not hasattr(self.scopes[scopeIndex].entityList[entityIndex], "arguments"):
201             self.scopes[scopeIndex].entityList[entityIndex].arguments = []
202         self.scopes[scopeIndex].entityList[entityIndex].arguments.append(newArgument)
203         return newArgument
204
205 def searchEntity(self, entityName):
206     for scope in self.scopes:
207         for entity in scope.entityList:
208             if entity.name == entityName:
209                 return entity, None
210     return None, "Entity not found."
211

```

Figure 41

Her methods `RecordStructure`

1. `__init__(self)`:
 - o Initializes an empty list of fields (scopes).
2. `addNewScope(self)`:
 - o Adds a new field to the list of fields.

- o Returns the new field.
- 3.**scopeDeletion(self, scopeIndex):**
 - o Deletes a field from the list of fields according to its index.
- 4.**addNewEntity(self, scopeIndex, entityName, entity_type, ** kwargs):**
 - o Adds a new entity to a specified field.
 - o entity_type may be 'Variable', 'Function' or 'TemporaryVariable'.
 - o Returns the new entity.
- 5.**addNewArgument(self, scopeIndex, entityIndex, argumentName, parMode, type):**
 - o Adds a new argument to a function within a specified scope and entity.
 - o Returns the new argument.
- 6.**searchEntity(self, entityName):**
 - o Searches for an entity by its name in all fields. Returns the entity and nesting level (or None if not found).
- 7.**printScopesToFile(self, o filename="cpy.sym"):**
 - Prints the fields and entities to a file. Creates a file with a reference to each field and the entities it contains.

```

212     def printScopesToFile(self, filename="cpy.sym"):
213         with open(filename, "w") as file:
214             file.write("-----SYMBOL TABLE-----\n")
215             for i, scope in enumerate(self.scopes, start=1):
216                 file.write(f"Scope {i-1}:\n")
217                 #file.write(f"Nesting Level: {scope.nestingLevel}\n")
218                 file.write("Entities:\n")
219                 for j, entity in enumerate(scope.entityList, start=1):
220                     file.write(f"\tEntity {j}:\n")
221                     file.write(f"\tName: {entity.name},")
222                     if isinstance(entity, self.Variable):
223                         file.write(f"\tOffset: {entity.offset}\n")
224                     elif isinstance(entity, self.Function):
225                         file.write(f"\tFrame Length: {entity.frameLength},")
226                         file.write(f"\tArguments:{entity.arguments}\n")
227
228                     elif isinstance(entity, self.TemporaryVariable):
229                         file.write(f"\tOffset: {entity.offset}\n")
230             file.write("\n")

```

Figure 42

7) Final code

For the symbol table, the Final class was created (Figures 43,44,5 & 46) which contains the helper functions of the slides and the handbook (instead of producing there are methods for almost every case). Also created were writeFunctionFinalCode(quadSubList) (generates final code according to the quads it accepts) and readQuadList(quad) (helper to generate the final code) (Figures 47,48,49,50,51 & 52) which help to writing the final code. Because fp does not exist in RISC-V, s0 was used in its place.

```
4  class Final:
5      def __init__(self, record_structure):
6          self.record_structure = record_structure
7          self.instructions = []
8
9      def gnlvcode(self, var_name, reg):
10         entity = self.record_structure.searchEntity(var_name)[0]
11         entity_level = self.record_structure.searchEntity(var_name)[1]
12         if entity is None:
13             print (var_name,"has not been initialized")
14             print("Running has failed")
15             exit()
16         current_level = len(self.record_structure.scopes)
17         levels_up = current_level - entity_level
18         if levels_up > 0:
19             self.instructions.append(f"\tlw {reg}, -4(sp)")
20         for _ in range(1, levels_up):
21             self.instructions.append(f"\tlw {reg}, -4({reg})")
22         self.instructions.append(f"\taddi {reg}, {reg}, {entity.offset}")
23
24     def loadvr(self, v, reg):
25         def increment_reg(reg):
26             prefix = reg[0]
27             num = int(reg[1:])
28             return f"{prefix}{num + 1}"
29
30         if str(v).isnumeric():
31             self.instructions.append(f"\tli {reg}, {v}")
32             return
33
34         entity_level = self.record_structure.searchEntity(v)[1]
35         entity = self.record_structure.searchEntity(v)[0]
36         if entity is None:
37             print (v,"has not been initialized")
38             print("Running has failed")
39             exit()
40
41         current_level = len(self.record_structure.scopes)
42
```

Figure 43

- `__init__(self, record_structure)`: This method is the constructor of the class. Accepts an object `record_structure`, which probably represents the structure of the source code. Initializes the fields `record_structure`, which refers to the structure of the source code, and `instructions`, which stores the execution commands.
- `gnlvcode(self, var_name, reg)`: This method generates code that returns the memory address of a variable name. It is mainly used to address the variable in an outer level of scope.
- `loadvr(self, v, reg)`: This method generates code that loads the value of a variable or a constant into a special register `reg`. `increment_reg(reg)` was created to give the next register
- `storerv(self, v, reg)`: This method generates code that stores the value from a special register `reg` to a variable `v`.

```

43     if entity_level == 1:
44         self.instructions.append(f"\tlw {reg}, -{entity.offset}(gp)")
45     elif entity_level == current_level:
46         if isinstance(entity, RecordStructure.Variable) or isinstance(entity, RecordStructure.TemporaryVariable):
47             self.instructions.append(f"\tlw {reg}, -{entity.offset}(sp)")
48         elif isinstance(entity, RecordStructure.RecordArgument) and entity.parMode == "CV":
49             self.instructions.append(f"\tlw {reg}, -{entity.offset}(sp)")
50             self.instructions.append(f"\tlw {reg}, 0({reg})")
51             next_reg = increment_reg(reg)
52             self.instructions.append(f"\tlw {next_reg}, 0({reg})")
53     elif entity_level < current_level:
54         self.gnlvcode(v, reg)
55         self.instructions.append(f"\tlw {reg}, 0({reg})")
56         next_reg = increment_reg(reg)
57         self.instructions.append(f"\tlw {next_reg}, 0({reg})")
58
59     def storerv(self, v, reg):
60         entity_level = self.record_structure.searchEntity(v)[1]
61         entity = self.record_structure.searchEntity(v)[0]
62         if entity is None:
63             print(v, "has not been initialized")
64             print("Running has failed")
65             exit()
66
67         current_level = len(self.record_structure.scopes)
68         if entity_level == current_level:
69             self.instructions.append(f"\tsv {reg}, -{entity.offset}(sp)")
70         elif entity_level < current_level:
71             if isinstance(entity_level, RecordStructure.RecordArgument) and entity.parMode == "CV":
72                 self.gnlvcode(v, reg)
73                 self.instructions.append(f"\tsv {reg}, 0({reg})")
74         elif entity_level == 1:
75             self.instructions.append(f"\tsv {reg}, -{entity.offset}(gp)")
76

```

Figure 44

- `end(self)`: This method generates code that terminates program execution.
- `callFun(self, funName)`: This method generates code that calls a function with the name `funName`.

- `returnToCaller(self)`: This method generates code that returns to the calling code from a function.
- `jump(self, jumpName)`: This method generates code that jumps to a tag named `jumpName`.

```

72     def end(self):
73         self.instructions.append("\tli a0, 0")
74         self.instructions.append("\tli a7, 93")
75         self.instructions.append("\tecall")
76
77     def callFun(self, funName):
78         self.instructions.append(f"\tjal ra, {funName}")
79
80     def retToCaller(self):
81         self.instructions.append("\tlw ra, 0(sp)")
82         self.instructions.append("\tjr ra")
83
84     def jump(self, jumpName):
85         self.instructions.append(f"\tj {jumpName}")
86
87     def label(self, labelName):
88         self.instructions.append(f"{labelName}:")
89
90     def move(self, r1, r2):
91         self.instructions.append(f"\tmv {r1}, {r2}")
92
93     def operations(self, r1, r2, r3, op):
94         if op == "+":
95             self.instructions.append(f"\tadd {r1}, {r2}, {r3}")
96         elif op == "-":
97             self.instructions.append(f"\tsub {r1}, {r2}, {r3}")
98         elif op == "*":
99             self.instructions.append(f"\tmul {r1}, {r2}, {r3}")
100        elif op == "/":
101            self.instructions.append(f"\tdiv {r1}, {r2}, {r3}")
102        elif op == "not":
103            self.instructions.append(f"\tnot {r1}, {r2}")
104        elif op == "or":
105            self.instructions.append(f"\tor {r1}, {r2}")
106        elif op == "and":
107            self.instructions.append(f"\tand {r1}, {r2}")
108

```

Figure 45

- `label(self, labelName)`: This method creates a label with the name `labelName`.
- `move(self, r1, r2)`: This method generates code that copies the value from a register `r2` to another register `r1`.

- `operations(self, r1, r2, r3, op)`: This method generates code for various operations (eg addition, subtraction) between the values located in three registers (`r1,r2,r3`).

```

108
109     def branch(self, r1, r2, label, con):
110         if con == "==":
111             self.instructions.append(f"\tbeq {r1}, {r2}, {label}")
112         elif con == "!=":
113             self.instructions.append(f"\tbne {r1}, {r2}, {label}")
114         elif con == ">=":
115             self.instructions.append(f"\tblt {r1}, {r2}, {label}")
116         elif con == "<":
117             self.instructions.append(f"\tbge {r1}, {r2}, {label}")
118
119     def write_instructions(self,s):
120         for instr in self.instructions:
121             s+=(instr + "\n")
122         self.instructions = []
123         return s
124

```

Figure 46

- `branch(self, r1, r2, label, con)`: This method generates code that programmatically implements program execution in case a certain condition is true or false.
- `write_instructions(self, s)`: This method creates a string which contains the execution commands that have been created and are stored in the field `instructions`

```

1325     def writeFunctionFinalCode(quadSubList):
1326         global finalCode
1327         global framelen
1328         branchRelOperators = ["=", ">", "<", "!=", ">=", "<="]
1329         arithmeticOperators = ["+", "-", "*", "/", "and", "not", "or"]
1330         index = 0
1331         while(index < len(quadSubList)):
1332             quad = quadSubList[index]
1333             currentLabel = getNextLabel()
1334             finalCode += currentLabel + ":\n"
1335             if(quad[0] == "begin_block" and quad[1] == "main"):
1336                 finalCode += "Lmain:\n"
1337
1338
1339             if(quad[0] == "begin_block"):
1340                 funcBeginLabels[quad[1]] = currentLabel
1341                 finalCode += "\tsw ra,0(sp)\n"
1342
1343             elif(quad[0] == "end_block"):
1344                 final.retToCaller()
1345                 finalCode = final.write_instructions(finalCode)
1346
1347             elif(quad[0] == ":="):
1348                 if(recordStructure.searchEntity(quad[1]) is not None):
1349                     final.loadvr(str(quad[1]), "t0")
1350                     finalCode = final.write_instructions(finalCode)
1351                 else:
1352                     final.gnlvcode(str(quad[1]), "t0")
1353                     finalCode = final.write_instructions(finalCode)
1354
1355                 final.storerv((quad[3]), "t0")
1356                 finalCode = final.write_instructions(finalCode)
1357
1358             elif(quad[0] in branchRelOperators):
1359                 final.loadvr(str(quad[1]), "t0")
1360                 finalCode = final.write_instructions(finalCode)
1361                 final.loadvr(str(quad[2]), "t1")
1362                 finalCode = final.write_instructions(finalCode)
1363                 final.branch("t0", "t1", "L"+str(quad[3]), str(quad[0]))
1364                 finalCode = final.write_instructions(finalCode)
1365

```

Figure 47

```

1366 elif(quad[0] == "jump"):
1367     finalCode += ("\tj L"+str(quad[3])+"\n")
1368
1369 elif(quad[0] == "ret"):
1370     finalCode += "\tlw t0 -8(sp)\n\tlw t1, -"+str(pos)+"(sp)\n\tsw t1, 0(t0)\n"
1371
1372 elif(quad[0] in arithmeticOperators):
1373     if(str(quad[1]).isnumeric()):
1374         if(quad[0] == "+" or quad[0] == "-"):
1375             finalCode += "\taddi t0, zero, "+str(quad[1])+"\n"
1376         elif(quad[0] == "*" or quad[0] == "/"):
1377             finalCode += "\tmuli t0, "+str(quad[1])+"\n"
1378         else:
1379             final.loadvr(str(quad[1]), "t0")
1380             finalCode = final.write_instructions(finalCode)
1381
1382     if(str(quad[2]).isnumeric()):
1383         if(quad[0] == "+" or quad[0] == "-"):
1384             finalCode += "\taddi t1, zero, "+str(quad[2])+"\n"
1385         elif(quad[0] == "*" or quad[0] == "/"):
1386             finalCode += "\tmuli t1, "+str(quad[2])+"\n"
1387         else:
1388             final.loadvr(str(quad[2]), "t1")
1389             finalCode = final.write_instructions(finalCode)
1390
1391     final.operations("t0", "t1", "t2", str(quad[0]))
1392     finalCode = final.write_instructions(finalCode)
1393     final.storerv(str(quad[3]), "t2")
1394     finalCode = final.write_instructions(finalCode)
1395

```

Figure 48

```

1396 elif(quad[0] == "mod"):
1397     if(recordStructure.searchEntity(quad[1]) is not None):
1398         final.loadvr(str(quad[1]), "t0")
1399         finalCode = final.write_instructions(finalCode)
1400     else:
1401         final.gnlvcode(str(quad[1]), "t0")
1402         finalCode = final.write_instructions(finalCode)
1403
1404     if(recordStructure.searchEntity(quad[2]) is not None):
1405         final.loadvr(str(quad[2]), "t1")
1406         finalCode = final.write_instructions(finalCode)
1407     else:
1408         final.gnlvcode(str(quad[2]), "t1")
1409         finalCode = final.write_instructions(finalCode)
1410
1411     final.loadvr(str(quad[3]), "t2")
1412     finalCode = final.write_instructions(finalCode)
1413     finalCode += "\trem t2, t0, t1\n"
1414     final.storerv(str(quad[3]), "t2")
1415     finalCode = final.write_instructions(finalCode)
1416
1417 elif(quad[0] == "inp"):

```

Figure 49


```

1413
1414     elif(quad[0] == "inp"):
1415         final.loadvr(str(quad[1]),"t0")
1416         finalCode = final.write_instructions(finalCode)
1417         finalCode+= "\tli a0, 0\n\tli a2, 1\n\tli a7,63\n\tecall\n"
1418         final.move("t0","a0")
1419         finalCode = final.write_instructions(finalCode)
1420
1421     elif(quad[0] == "out"):
1422         final.loadvr(str(quad[1]),"t0")
1423         finalCode = final.write_instructions(finalCode)
1424         finalCode+= "\tlw a0, 0(t0)\n\tli a7,1\n\tecall\n"
1425
1426     elif(quad[0] == "halt"):
1427         final.end()
1428         finalCode = final.write_instructions(finalCode)
1429
1430     elif(quad[0] == "par"):
1431         finalCode+= "\taddi s0,sp,"+str(framelen)+"\n#### opou fp s0
1432         while(quad[0] == "par" and quad[2]!="ret"):
1433             entity,entity_level = recordStructure.searchEntity(quad[1])
1434             offset = entity.offset
1435             finalCode += getNextLabel()+":\n"
1436             final.loadvr(str(quad[1]),"t0")
1437             finalCode = final.write_instructions(finalCode)
1438             finalCode+= "\tsw t0, -"+str(offset)+"(s0)\n####opou fp s0
1439             index+=1
1440             quad = quadSubList[index]
1441
1442
1443             finalCode += getNextLabel()+":\n"
1444             finalCode+= "\taddi t0, sp, -"+str(offset)+"\n"
1445             finalCode+= "\tsw t0, -8(s0)\n####opou fp s0
1446             index+=1
1447             quad = quadSubList[index]
1448             finalCode += getNextLabel()+":\n"
1449             label = funcBeginLabels[quad[1]]
1450             final.callFun(label)
1451             finalCode = final.write_instructions(finalCode)
1452         index+=1

```

Figure 52

```

1454
1455     def readQuadList(quad):
1456         funcName = quad[1]
1457         endIndex = quadList.index(quad)
1458         beginIndex = quadList.index(["begin_block",funcName,"_", "_"])
1459         writeFunctionFinalCode(quadList[beginIndex:endIndex+1])
1460

```

Figure 52